Section 5: I/O and Deadlock

CS162

July 10, 2019

Contents

1 Vocabulary **2**

2 Problems **3**

2.1 Files . . . 3

2.1.1 Files vs File Descriptor . . . 3

2.1.2 Quick practice with write and seek . . . 3

2.2 Dup and Dup2 . . . 3

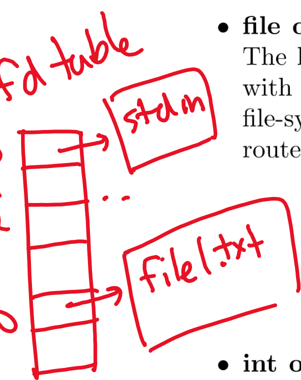
2.2.1 Warmup . . . 3

2.2.2 Redirection: executing a process after dup2 . . . 4

2.2.3 Redirecting in a new process . . . 5

2.3 Banker's Algorithm . . . 6

1 Vocabulary

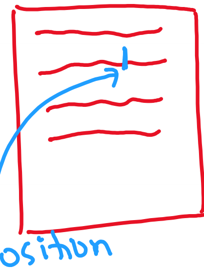


- **file descriptors** - File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an open call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Using file descriptors, a process's read or write calls are routed to the correct place by the kernel. When your program starts you have 3 file descriptors.

File Descriptor	File
0	stdin
1	stdout
2	stderr

- **int open(const char *path, int flags)** - open is a system call that is used to open a new file and obtain its file descriptor. Initially the offset is 0.

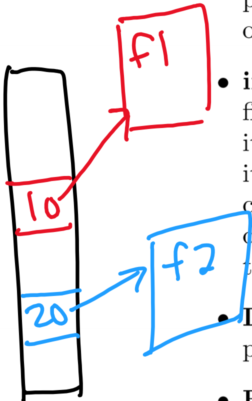
- **size_t read(int fd, void *buf, size_t count)** - read is a system call used to read `count` bytes of data into a buffer starting from the file offset. The file offset is incremented by the number of bytes read.



- **size_t write(int fd, const void *buf, size_t count)** - write is a system call that is used to write up to `count` bytes of data from a buffer to the file offset position. The file offset is incremented by the number of bytes written.

- **size_t lseek(int fd, off_t offset, int whence)** - lseek is a system call that allows you to move the offset of a file. There are three options for whence
  - SEEK_SET - The offset is set to `offset`.
  - SEEK_CUR - The offset is set to current_offset + `offset`
  - SEEK_END - The offset is set to the size of the file + `offset`

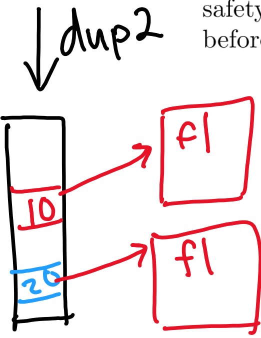
- **int dup(int oldfd)** - creates an alias for the provided file descriptor and returns the new fd value. dup always uses the smallest available file descriptor. Thus, if we called dup first thing in our program, it would use file descriptor 3 (0, 1, and 2 are already signed to stdin, stdout, stderr). The old and new file descriptors refer to the same open file description and may be used interchangeably.



- **int dup2(int oldfd, int newfd)** - dup2 is a system call similar to dup. It duplicates the `oldfd` file descriptor, this time using `newfd` instead of the lowest available number. If newfd was open, it closed before being reused. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the file descriptor, performing the redirection in one elegant command. For example, if you wanted to redirect standard output to a file, then you would call dup2, providing the open file descriptor for the file as the first input and 1 (standard output) as the second.

- **Deadlock** - Situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

- **Banker's Algorithm** - A resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, before deciding whether allocation should be allowed to continue.



2 Problems

2.1 Files

2.1.1 Files vs File Descriptor

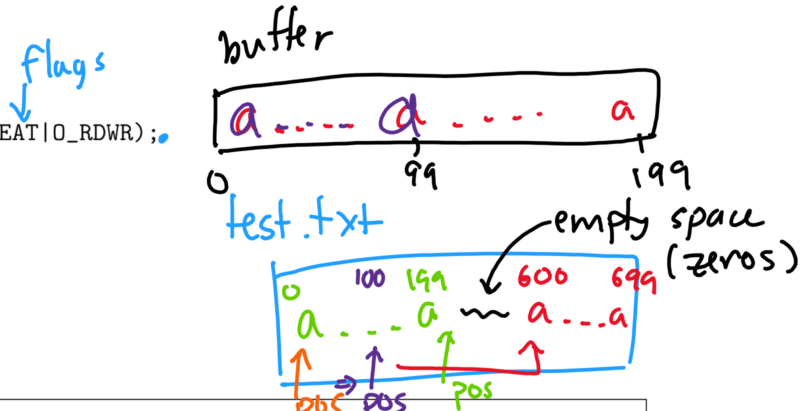
What's the difference between `fopen` and `open`?

fopen calls open (adds functionality around open) buffering, more methods

2.1.2 Quick practice with write and seek

What will the test.txt file look like after I run this program? (Hint: if you write at an offset past the end of file, the bytes inbetween the end of the file and the offset will be set to 0.)

```
int main() {
    char buffer[200];
    memset(buffer, 'a', 200);
    int fd = open("test.txt", O_CREAT|O_RDWR);
    write(fd, buffer, 200);
    lseek(fd, 0, SEEK_SET);
    read(fd, buffer, 100);
    lseek(fd, 500, SEEK_CUR);
    write(fd, buffer, 100);
}
```



a from 0-200, 0 from 200-600, a from 600-700

2.2 Dup and Dup2

2.2.1 Warmup

What does C print in the following code?

```
int main(int argc, char **argv)
{
    int pid, status;
    int newfd;

    if ((newfd = open("output_file.txt", O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        exit(1);
    }
    printf("The last digit of pi is...");
    dup2(newfd, 1);
    printf("five\n");
    exit(0);
}
```

prints to terminal
stdout
prints to output_file.txt

2.2.2 Redirection: executing a process after dup2

Describe what happens, and what the output will be.

```
int
main(int argc, char **argv)
{
    int pid, status;
    int newfd;
    char *cmd[] = { "/bin/ls", "-al", "/", 0 };

    if (argc != 2) {
        fprintf(stderr, "usage: %s output_file\n", argv[0]);
        exit(1);
    }
    if ((newfd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        perror(argv[1]);    /* open failed */
        exit(1);
    }
    printf("writing output of the command %s to \"%s\"\n", cmd[0], argv[1]);
    dup2(newfd, 1);
    execvp(cmd[0], cmd);
    perror(cmd[0]);     /* execvp failed */

    exit(1);
}
```

2.2.3 Redirecting in a new process

Modify the above code such that the result of ls -al is written to the file specified by the input argument and immediately after "all done" is printed to the terminal. (Hint: you'll need to use fork and wait.)

doing in [illegible] above [illegible]

Conservative output:
safe ⇒ for sure no deadlock
unsafe ⇒ may or may not have deadlock

2.3 Banker's Algorithm

- pretend to give out resources ⇒ see if we can still finish execution

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total		
A	B	C
7	8	9

	Current			Max		
T/R	A	B	C	A	B	C
T1	0	2	2	4	3	3
T2	2	2	1	3	6	9
T3	3	0	4	3	1	5
T4	1	3	1	3	3	4

	A	B	C
T1 needs:	4	1	1
T2 " :	1	4	8
T3 " :	0	1	1
T4 " :	2	0	3

Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

Time	
0	Avail: 7 − 0 − 2 − 3 − 1 = 1 8 − 2 − 2 − 0 − 3 = 1 9 − 2 − 1 − 4 − 1 = 1
1	Run T3 → T3 finishes (free resources) → 1 + 3 = 4 1 + 0 = 1 1 + 4 = 5
2	Run T1 → 4 + 0 = 4 1 + 2 = 3 5 + 2 = 7
3	Run T4 → 4 + 1 = 5 3 + 3 = 6 7 + 1 = 8 → Run T2

⇒ SAFE

Repeat the previous question if the total number of C instances is 8 instead of 9.

0 | Avail: 1 1 0

→ no thread can be safely scheduled

⇒ Unsafe

deadlock?
~ maybe not. Bankers assumes all threads need Max resources all @ once to complete (might not be true when actually run).